UPC-IO: A Parallel I/O API for UPC

V1.0pre10

Tarek El-Ghazawi
François Cantonnet
Proshanta Saha
The George Washington University
801 22nd Street NW • Suite 607
Washington, DC 20052, USA
{tarek, fcantonn, sahap}@gwu.edu

Rajeev Thakur
Rob Ross
Mathematics and Computer Science Division
Argonne National Laboratory
9700 S. Cass Avenue
Argonne, IL 60439, USA
{thakur, rross}@mcs.anl.gov

Dan Bonachea
Dept. of Computer Science
University of California, Berkeley
Berkeley, CA 94720, USA
bonachea@cs.berkeley.edu

October 28, 2003

UPC-IO:
A Parallel I/O API for UPC

Table of Contents

3 Terms, definitions and symbols

3.1 Collective

1 As per the UPC Language Specifications Document [1], the term collective is defined as "a requirement placed on some language operations which constrains invocations of such operations to be matched across all threads. The behavior of collective operation is undefined unless all threads execute the same sequence of collective operations".

3.2 List Based File Access

1 File accesses done using explicit offsets and sizes of data. Non-contiguous accesses may be performed using lists of explicit offsets and lengths in the file.

3.3 File Pointer Based Access

1 File accesses are done using either private or shared file pointers, where private file pointers provide a means for each thread to independently control its access of the file and shared file pointers provide a means for all the threads to access the file synchronously.

3.4 Synchronous I/O Call

1 I/O calls which block and wait until the corresponding I/O operation is completed.

3.5 Asynchronous I/O Call

1 I/O calls which start an I/O operation and return immediately, and must later be completed using a synchronization function. Only one outstanding asynchronous operation is allowed on a UPC-IO file handle at any given time.

3.6 Consistency Semantics

1 Consistency semantics define when the data written to a file by a thread is visible to other threads. The consistency semantics also define the outcome in the case of overlapping reads into a shared buffer in memory when using private file pointers or list I/O functions.

3.7 Atomicity Semantics

1 Atomicity semantics define the outcome of operations in which multiple threads write concurrently to a file and some of the writes overlap each other.

7 Library

7.3 UPC Parallel I/O <upc_io.h>

1 This subsection provides the UPC parallel extensions of section 7.19 in [9]. All the characteristics of library functions described in section 7.1.4 in [9] apply to these as well.

Common Constraints

1 All UPC-IO functions are collective and must be called by all threads. (See Section 3.6 of the UPC Specification [1] for the definition of collective).

2 If a program calls `exit`, `upc_global_exit`, or returns from `main` with a UPC file still open, the file will automatically be closed at program termination, and the effect will be equivalent to `upc_all_close` being implicitly called on the file.

3 If a program attempts to read past the end of a file, the read function will read data up to the end of file and return the number of bytes actually read, which may be less than the amount requested.

4 Writing past the end of a file increases the file size.

5 If a program seeks to a location past the end of a file and writes starting from that location, the data in the intermediate (unwritten) portion of the file is undefined. For example, if a program opens a new file (of size 0 bytes), seeks to offset 1024 and writes some data beginning from that offset, the data at offsets 0–1023 is undefined. Seeking past the end of file and performing a write causes the current file size to immediately be extended up to the end of the write. However, just seeking past the end of file or attempting to read past the end of file, without a write, does not extend the file size.

6 All "`shared void *`" pointers passed to the I/O functions (as function arguments or indirectly through the list I/O arguments) are treated as if they had a phase field of zero (that is, the input phase is ignored).

7 An implementation is permitted to begin performing a collective UPC-IO operation when the first thread reaches the collective operation, and the operation is not guaranteed to be complete until the last thread leaves the collective operation. In other words, collective does not imply barrier semantics (The exact behavior of UPC collective operations is currently being defined in the UPC Collective Specifications [10]).

8 The arguments to all UPC-IO functions are single-valued (must have the same value on all threads) except where explicitly noted otherwise in the function description.

9 UPC-IO, by default, supports weak consistency and atomicity semantics. The default (weak) semantics are as follows. The data written by a thread is only guaranteed to be

visible to another thread after all threads have collectively closed or synchronized the file.

Writes from a given thread are always guaranteed to be visible to subsequent reads by the *same* thread, even without an intervening call to collectively close or synchronize the file.

Byte-level data consistency is supported.

If concurrent writes from multiple threads overlap in the file, the resulting data in the overlapping region is undefined with the weak consistency and atomicity semantics

When reading data being concurrently written by another thread, the data that gets read is undefined with the weak consistency and atomicity semantics.

Overlapping reads into a shared buffer in memory using private file pointers or list I/O functions leads to undefined data being read under the weak consistency and atomicity semantics.

A given file can not (should not) be opened at same time by the POSIX I/O and UPC-IO libraries.

Except where otherwise noted, all UPC-IO functions return NON-single-valued errors; that is, the occurrence of an error need only be reported to at least one thread, and the errno value reported to each such thread may differ. If an error is reported to ANY thread, the position of ALL file pointers for the relevant file handle becomes undefined.

7.3.0 Background

7.3.0.1 File Accessing and File Pointers

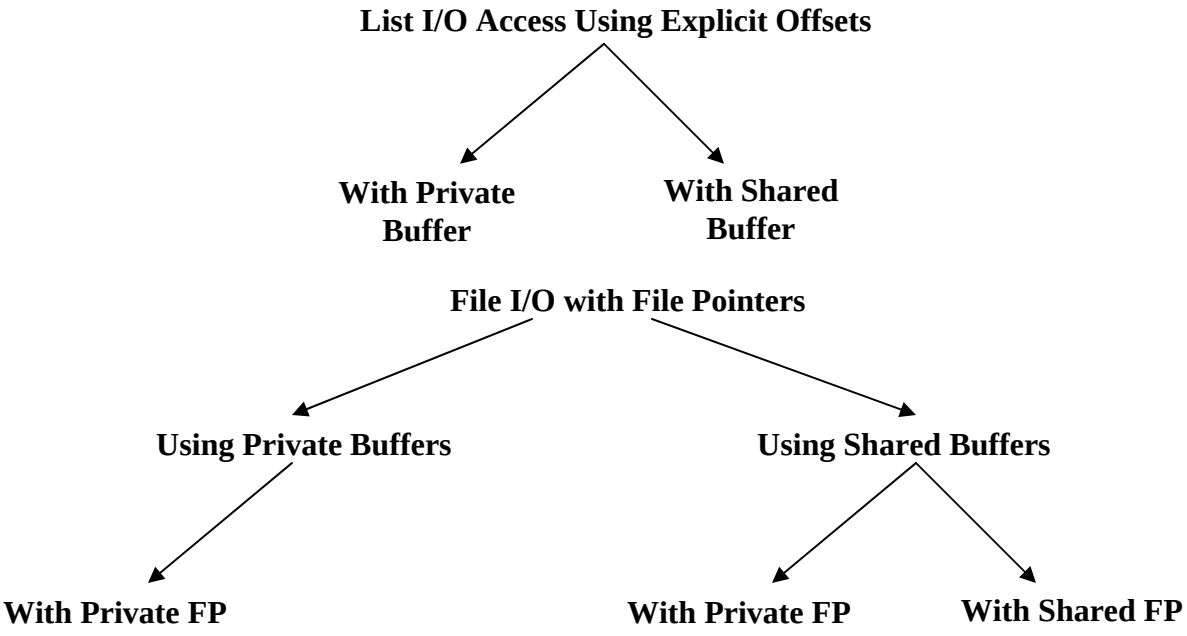

Figure 1. UPC-IO File Access Methods

Collective UPC-IO accesses can be done in and out of shared and private buffers, thus private and shared reads and writes are generally supported. In each of these cases, file pointers could be either shared or private. File pointer modes are specified by passing a flag to the collective upc_all_open function. When a file is opened with the shared file pointer flag, all threads share a common file pointer. When a file is opened with the private file pointer flag, each thread gets its own file pointer.

Note that in UPC-IO, shared file pointers cannot be used in conjunction with private buffers. UPC-IO also provides list file accesses by specifying explicit offsets and sizes of data that is to be accessed. List IO can also be used with either private buffers or shared buffers.

Examples 1-3 and their associated figures, Figures 2-4, give typical instances of UPC-IO usage.

Example 1:

```
char buffer[10];  // and assuming a total of 4 THREADS
upc_file_t *fd;

fd = upc_all_open( “file”, UPC_RDONLY | UPC_PRIVATE_FP, 0, NULL );
upc_all_seek( fd, 5*MYTHREAD, UPC_SEEK_SET );
upc_all_read_private( fd, buffer, 10 );
upc_all_close(fd);
```

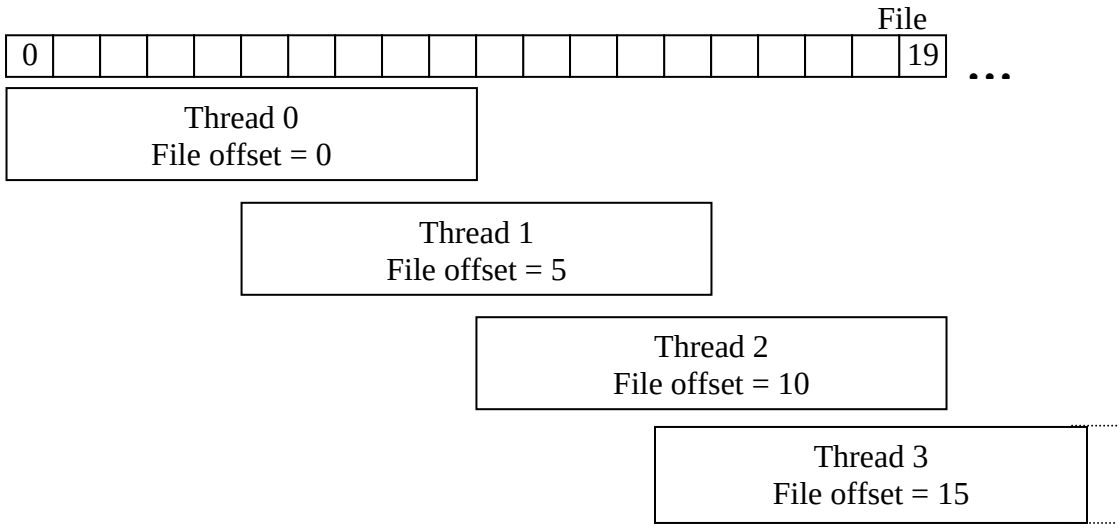

Figure 2. Collective read into private can provide Canonical file-view

Figure 2 represents the previous example which considers a collective read operation. Each thread reads from a particular thread-specific offset, a block of data into a private buffer. Figure 3 illustrates how the file is viewed in a case of example 2. The data read is stored into a shared buffer, having a block size of 5 elements. The user selects the type of file pointer at file-open time. The user can select either private file pointers by passing the flag `UPC_PRIVATE_FP` to the function `upc_all_open`, or the shared file pointer by passing the flag `UPC_SHARED_FP` to `upc_all_open.`

Example 2:

```
shared [5] char buffer[20];  // and assuming a total of 4 static THREADS
upc_file_t *fd;

fd = upc_all_open( “file”, UPC_RDONLY | UPC_SHARED_FP, 0, NULL );
upc_all_read_shared( fd, buffer, 5, 20 );
```

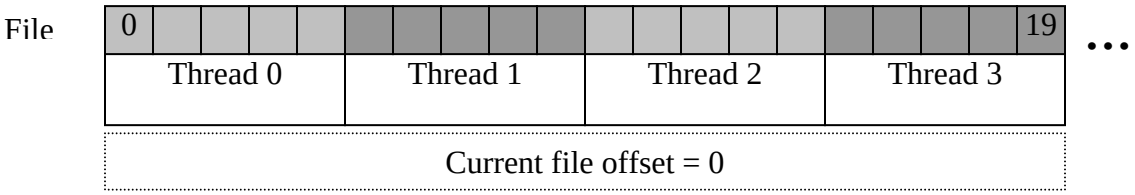

Figure 3. Collective read into a blocked shared buffer can provide a partitioned file-view

The list I/O functions allow the user to specify noncontiguous accesses both in memory and file in the form of lists of explicit offsets and lengths in the file and explicit address and lengths in memory. None of the file pointers are used or updated in this case. An example of a program that uses list I/O is Example 3. The resulting data transfer is illustrated in Figure 8.

Example 3:

```
upc_private_memvec_t memvec[2];
upc_filevec_t filevec[2];
upc_file_t *fd;
char buffer[12];

fd = upc_all_open( “file”, UPC_RDONLY| UPC_PRIVATE_FP, 0, NULL );
mem_vec[0].baseaddr = &buffer[0];
mem_vec[0].len = 4;
mem_vec[1].baseaddr = &buffer[7];
mem_vec[1].len = 3;
file_vec[0].offset = MYTHREAD*5;
file_vec[0].len = 2;
file_vec[1].offset = 10+MYTHREAD*5;
file_vec[1].len = 5;

upc_all_read_list_private( fd, 2, &mem_list, 2, &file_list );
```

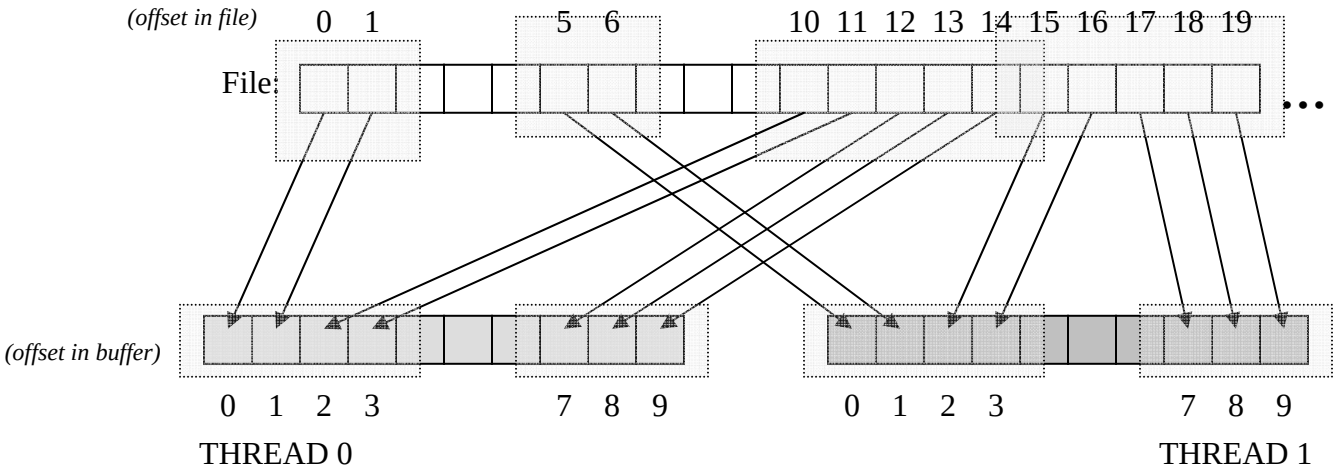

Figure 4. List I/O read of noncontiguous parts of a file to private noncontiguous buffers

7.3.0.2 Synchronous and Asynchronous I/O

I/O operations can be synchronous (blocking) or asynchronous (non-blocking). While synchronous calls are quite simple and easy to use from a programming point of view, asynchronous operations allow the overlapping of computation and I/O to achieve improved performance. Synchronous calls block and wait until the corresponding I/O operation is completed. On the other hand, an asynchronous call starts an I/O operation and returns immediately. Thus, the executing process can turn its attention to other processing needs while the I/O is progressing.

UPC-IO supports both synchronous and asynchronous I/O functions. The asynchronous I/O functions have the same syntax as their synchronous counterparts, with the addition of "`async`" in their names. The asynchronous I/O functions have the restriction that only one (collective) asynchronous operation can be active at a time on a given file handle. That is, an asynchronous I/O function must be completed by calling `upc_all_async_test` or `upc_all_async_wait` before another asynchronous I/O function can be called on the same file handle. This restriction is similar to the restriction MPI-IO has on split-collective I/O functions: only one split collective operation can be outstanding on an MPI-IO file handle at any time.

7.3.0.3 Consistency and Atomicity Semantics

Let us first define what we mean by the terms consistency semantics and atomicity semantics. The consistency semantics define when the data written to a file by a thread is visible to other threads. The atomicity semantics define the outcome of operations in which multiple threads write concurrently to a file and some of the writes overlap each other. For performance reasons, UPC-IO, by default, supports weak consistency and atomicity semantics. The user can select stronger semantics by passing the flags `UPC_STRICT` to `upc_all_open`.

The default (weak) semantics are as follows. The data written by a thread is only guaranteed to be visible to another thread after all threads have called `upc_all_close` or `upc_all_file_sync`. (Note that the data *may* be visible to other threads before the call to `upc_all_close` or `upc_all_file_sync` and that the data may become visible to different threads at different times.) Writes from a given thread are always guaranteed to be visible to subsequent reads by the *same* thread, even without an intervening call to `upc_all_close` or `upc_all_file_sync.` Byte-level data consistency is supported. That is, for example, if thread 0 writes one byte at offset 0 in the file and thread 1 writes one byte at offset 1 in the file, the data from both threads will get written to the file. If concurrent writes from multiple threads overlap in the file, the resulting data in the overlapping region is undefined. Similarly, if one thread tries to read the data being concurrently written by another thread, the data that gets read is undefined. Concurrent in this context means any two read/write operations to the same file handle with no intervening calls to `upc_all_file_sync` or `upc_all_close`.

For the functions that read into or write from a shared buffer using a shared file pointer, the weak consistency semantics are defined as follows. Each call to `upc_all_[async_]{read,write}_shared` with a shared file pointer behaves as if the read/write operations were performed by a single, distinct, anonymous thread which is different from any compute thread (and different for each operation). In other words, NO reads are guaranteed to see the results of writes using the shared file pointer until after a close or sync under the default weak consistency semantics.

By passing the UPC_STRICT flag to upc_all_open, the user selects strong consistency and atomicity semantics. In this case, the data written by a thread is visible to other threads as soon as the write on the calling thread returns. In the case of writes from multiple threads to overlapping regions in the file, the result would be as if the individual write function from each thread occurred atomically in some (unspecified) order. Overlapping writes to a file in a single (list I/O) write function on a single thread are not permitted (see Section 7.3.5).

The consistency semantics also define the outcome in the case of overlapping reads into a shared buffer in memory when using private file pointers or list I/O functions. By default, the data in the overlapping space is undefined. If the user selects the UPC_STRICT mode, the result would be as if the individual read functions from each thread occurred atomically in some (unspecified) order. Overlapping reads into memory buffers in a single (list I/O) read function on a single thread are not permitted (see Section 7.3.5).

Note that in the strict case, atomicity is guaranteed at the UPC-IO function level. The entire operation specified by a single function is performed atomically, regardless of whether it represents a single, contiguous read/write or multiple noncontiguous reads or writes as in a list I/O function.

Example

Consider the following example in which three threads write data concurrently, each with a single list I/O function. The numbers indicate file offsets and brackets indicate the boundaries of a listed vector. Each thread writes its own thread id as the data values:

```
thread 0:     {1  2  3}   {5  6  7  8}
thread 1:  {0  1  2}{3  4  5}
thread 2:              {4  5  6}   {8  9  10  11}
```

With the default weak semantics, the results in the overlapping locations are undefined. Therefore, the result in the file would be the following, where x represents undefined data.

```
File:       1  x  x  x  x  x  x  0  x  2  2   2
```

That is, the data from thread 1 is written at location 0, the data from thread 0 is written at location 7, and the data from thread 2 is written at locations 9, 10, and 11, because none of these locations had overlapping writes. All other locations had overlapping writes, and consequently, the result at those locations is undefined.

If the file were opened with the UPC_STRICT flag, strict semantics would be in effect. The result, then, would depend on the order in which the writes from the three threads actually occurred. Since six different orderings are possible, there can be six outcomes. Let us assume, for example, that the ordering was the write from thread 0, followed by

the write from thread 2, and then the write from thread 1. The (list I/O) write from each thread happens atomically. Therefore, for this ordering, the result would be:

```
File:       1  1  1  1  1  1  2  0  2  2  2   2
```

We note that if instead of using a single list I/O function each thread used a separate function to write each contiguous portion, there would be six write functions, two from each thread, and the atomicity would be at the granularity of the write operation specified by each of those functions.

7.3.0.4 File Interoperability

UPC-IO does not specify how an implementation may store the data in a file on the storage device. Accordingly, whether a file created by UPC-IO can be directly accessed by using C/POSIX I/O functions or not is implementation-defined. However, the UPC-IO implementation must specify how the user can retrieve the file from the storage system as a linear sequence of bytes and vice versa. Similarly, the implementation must specify how familiar operations, such as the equivalent of POSIX `ls`, `cp`, `rm`, and `mv` can be performed on the file.

7.3.1 Predefined Types

7.3.1.1 The `upc_off_t` type

Synopsis

```
1  #include <upc_io.h>

   upc_off_t myOffset;
```

Description

1 A signed integer that is capable of representing the size of the largest file supported by the implementation:
`upc_off_t`

7.3.1.2 The `upc_file_t` type

Synopsis

```
1  #include <upc_io.h>

   upc_file_t *myFile;
```

Description

1 An opaque shared datatype of incomplete type (as defined in section 6.2.5 of [9]) that represents an open file handle:
`upc_file_t`

Constraints

1 `upc_file_t` objects are always manipulated via a pointer (that is, `upc_file_t *`).

Advice to implementors

1 As `upc_file_t` is a shared datatype, it is legal to pass a (`upc_file_t *`) across threads, and two pointers to `upc_file_t` that reference the same logical file handle will always compare equal.

2 The definition of `upc_file_t` does not restrict the implementation to store all its metadata with affinity to one thread. Each thread can still have local access to its metadata. For example, below is a simple approach an implementation could use:

```
/* for a POSIX-based implementation */
typedef int my_internal_filehandle_t;

#ifdef _UPC_INTERNAL
  typedef struct _priv_upc_file_t  {
    my_internal_filehandle_t fd;
    ... other metadata ...
  } priv_upc_file_t;
#else
  struct _priv_upc_file_t;
#endif

typedef shared struct _priv_upc_file_t upc_file_t;

upc_file_t *upc_all_open(...) {

   upc_file_t *handles =
            upc_all_alloc(THREADS, sizeof(upc_file_t));

   /* get my handle */
   upc_file_t *myhandle = &(handles[MYTHREAD]);

   /* cast to a private pointer */
   priv_upc_file_t* myprivhandle =
(priv_upc_file_t*)myhandle;
```

```
    /* setup my metadata using private pointer */
    myprivhandle->fd = open(...);

    ...

    return handles;
}
```

The basic idea is that the “handle” exposed to the user actually points to a cyclic, distributed array. As a result, each thread has easy, local access to its own internal handle metadata with no communication, while maintaining the property that the handle that UPC-IO exposes to the client is a single-valued pointer-to-shared. An additional advantage is that a thread can directly access the metadata for other threads, which may occasionally be desirable in the implementation.

7.3.1.3 The `upc_private_memvec_t` type

Synopsis

```
1  #include <upc_io.h>

   upc_private_memvec_t myPrivateMemoryVector;
```

Description

1 `upc_private_memvec_t` is defined as follows:

```
typedef struct {
  void *baseaddr;
  size_t len;
} upc_private_memvec_t;
```

`baseaddr` and `len` specify a contiguous memory region in terms of the base address and length in bytes. `len` may be zero, in which case that entry is ignored.

7.3.1.4 The `upc_shared_memvec_t` type

Synopsis

```
1  #include <upc_io.h>

   upc_shared_memvec_t mySharedMemoryVector;
```

Description

1 `upc_shared_memvec_t` is defined as follows:

```
typedef struct {
  shared void *baseaddr;
  size_t blocksize;
  size_t len;
} upc_shared_memvec_t;
```

`baseaddr` and `len` specify a shared memory region in terms of the base address and length in bytes. `len` may be zero, in which case that entry is ignored. `blocksize` is the block size of the shared buffer in bytes. A `blocksize` of 0 indicates an indefinite blocking factor.

7.3.1.5 The `upc_filevec_t` type

Synopsis

1
```
#include <upc_io.h>

upc_filevec_t myFileVector;
```

Description

1 `upc_filevec_t` is defined as follows:

```
typedef struct {
  upc_off_t offset;
  size_t len;
} upc_filevec_t;
```

`offset` and `len` specify a contiguous region in the file in terms of the starting offset in the file in bytes and the length.

7.3.1.6 The `upc_hint_t` type

Synopsis

1
```
#include <upc_io.h>

upc_hint_t myHint;
```

Description

1 `upc_hint_t` is defined as follows:

```
typedef struct {
  const char *key;
  const char *value;
} upc_hint_t;
```

2 UPC-IO supports a number of predefined hints. An implementation is free to support additional hints. An implementation is free to ignore any hint provided by the user. Implementations should *silently* ignore any hints they do not support or recognize. The predefined hints and their meanings are defined below. An implementation is not required to interpret these key values, but if it does interpret the key value, it must provide the functionality described. For each hint name introduced, we describe the type of the hint value and its meaning. All hints are single-valued.

`access_style` (comma-separated list of strings): indicates the manner in which the file is expected to be accessed. The hint value is a comma-separated list of any the following: `read_once`, `write_once`, `read_mostly`, `write_mostly`, `sequential`, and `random`. Passing such a hint does not place any constraints on how the file may actually be accessed by the program, although accessing the file in a way that is different from the specified hint may result in lower performance.

`collective_buffering` (boolean): specifies whether the application may benefit from collective buffering optimizations. Legal values for this key are "`true`" and "`false`". Collective buffering parameters can be further directed via additional hints: `cb_buffer_size`, and `cb_nodes`.

`cb_buffer_size` (integer): specifies the total buffer space that the implementation can use on each thread for collective buffering.

`cb_nodes` (integer): specifies the number of target threads or I/O nodes to be used for collective buffering.

`file_perm` (string): specifies the file permissions to use for file creation. The set of legal values for this key is implementation dependent.

`io_node_list` (comma separated list of strings): specifies the list of I/O devices that should be used to store the file and is only relevant when the file is created.

`nb_proc` (integer): specifies the number of threads that will typically be used to run programs that access this file and is only relevant when the file is created.

`striping_factor` (integer): specifies the number of I/O devices that the file should be striped across and is relevant only when the file is created.

`start_io_device` (integer): specifies the number of the first I/O device from which to start striping the file and is relevant only when the file is created.

`striping_unit` (integer): specifies the striping unit to be used for the file. The striping unit is the amount of consecutive data assigned to one I/O device before progressing to the next device, when striping across a number of devices. It is expressed in bytes. This hint is relevant only when the file is created.

7.3.2 UPC File Operations

Common Constraints

1 All UPC-IO functions are collective and must be called by all threads. (See [10] for a definitive definition of collective).

2 The arguments to all UPC-IO functions are single-valued (must have the same value on threads) except where explicitly noted otherwise in the function description.

3 If the type of file pointer is not provided at file open, an error will be returned.

4 When a file is opened with a private file pointer, each thread will get its own file pointer and advances through the file at its own pace.

5 When a shared file pointer is used, all threads positioned in the file will be aligned with that shared file pointer.

6 Shared file pointers cannot be used in conjunction with private buffers.

7 No function in this section (7.3.2) may be called while an asynchronous operation is pending on the file handle.

7.3.2.1 The `upc_all_open` function

Synopsis

```
#include <upc.h>
#include <upc_io.h>

    upc_file_t *upc_all_open(const char *fname,
                             int flags,
                             size_t numhints,
                             upc_hint_t const *hints)
```

Description

1 `upc_all_open` opens the file identified by the filename `fname` for input/output operations.

2 The `flags` parameter specifies the access mode. The valid flags and their meanings are listed below. Of these flags, exactly one of `UPC_RDONLY`, `UPC_WRONLY`, or `UPC_RDWR`, and one of UPC_SHARED_FP or UPC_PRIVATE_FP, must be used. Other flags are optional. Multiple flags can be passed by using the bitwise OR operator (`|`).

`UPC_RDONLY`	Open the file in read-only mode
`UPC_WRONLY`	Open the file in write-only mode
`UPC_RDWR`	Open the file in read/write mode
`UPC_PRIVATE_FP`	Use a private file pointer for all file accesses (other than list I/O)
`UPC_SHARED_FP`	Use the shared file pointer for all file accesses (other than list I/O)
`UPC_APPEND`	Set the *initial* position of the file pointer to end of file. (The file pointer is not moved to the end of file after each read/write.)
`UPC_CREATE`	Create the file if it does not already exist
`UPC_EXCL`	Used in conjunction with `UPC_CREATE`. The open will fail if the file already exists.
`UPC_STRICT`	Set strict consistency and atomicity semantics
`UPC_TRUNC`	Open the file and truncate it to zero length
`UPC_DELETE_ON_CLOSE`	Delete the file on close

3 The `UPC_SHARED_FP` flag specifies that all accesses (except for the list I/O operations) will use the shared file pointer. The `UPC_PRIVATE_FP` flag specifies that all accesses will use private file pointers (except for the list I/O operations). Either `UPC_SHARED_FP` or `UPC_PRIVATE_FP` must be specified or `upc_all_open` will return an error.

4 The `UPC_STRICT` flag specifies strict consistency and atomicity semantics. The data written by a thread is visible to other threads as soon as the write on the calling thread returns. In the case of writes from multiple threads to overlapping regions in the file, the result would be as if the individual write function from each thread occurred atomically in some (unspecified) order. In the case of overlapping reads into a shared buffer in memory when using private file pointers or list I/O functions, the result would be as if the individual read functions from each thread occurred atomically in some (unspecified) order.

5 If the flag `UPC_STRICT` is not specified, weak semantics are provided. The data written by a thread is only guaranteed to be visible to another thread after all

threads have called `upc_all_close` or `upc_all_file_sync`. (Note that the data *may* be visible to other threads before the call to upc_all_close or upc_all_file_sync and that the data may become visible to different threads at different times.) Writes from a given thread are always guaranteed to be visible to subsequent reads by the *same* thread, even without an intervening call to `upc_all_close` or `upc_all_file_sync`. Byte-level data consistency is supported. For the purposes of atomicity and consistency semantics, each call to `upc_all_[async_]{read,write}_shared` with a shared file pointer behaves as if the read/write operations were performed by a single, distinct, anonymous thread which is different from any compute thread (and different for each operation)."[*]

6 Hints can be passed to the UPC-IO library as an array of key-value pairs[†] of strings. `numhints` specifies the number of hints in the `hints` array; if `numhints` is zero, the `hints` pointer is ignored. The user can free the `hints` array as soon as the open call returns. Each element of the `hints` array is of type `upc_hint_t`.

7 A file on the storage device is in the *open* state from the beginning of an open call to the end of the matching close call. It is erroneous to have the same file *open* simultaneously with two `upc_all_open` calls, or with a `upc_all_open` call and a POSIX/C `open` or `fopen` call.

8 The user is responsible for ensuring that the file referenced by the `fname` argument refers to a single file. The actual argument passed on each thread may be different because the file name spaces may be different on different threads, but they must all refer to the same logical UPC-IO file.

9 On success, the function returns a pointer to a file handle that can be used to perform other operations on the file. If an error occurs, the function returns `NULL` and sets `errno` appropriately.

7.3.2.2 The `upc_all_close` function

Synopsis

```
#include <upc.h>
#include <upc_io.h>
```

[*] In other words, NO reads are guaranteed to see the results of writes using the shared file pointer until after a close or sync when `UPC_STRICT` is not specified.

[†] The contents of the key/value pairs passed by all the threads must be single valued.

```
int upc_all_close(upc_file_t *fd)
```

Description

1 `upc_all_close` executes an implicit upc_all_file_sync and then closes the file associated with `fd`.

2 The function returns 0 on success. If `fd` is not valid or if an outstanding asynchronous operation has not been completed, the function returns –1 and sets `errno` appropriately.

3 After a file has been closed with `upc_all_close`, the file can legally be opened and the data in it can be accessed by using regular C/POSIX I/O calls.

7.3.2.3 The `upc_all_file_sync` function

Synopsis

```
#include <upc.h>
#include <upc_io.h>

    int upc_all_file_sync(upc_file_t *fd)
```

Description

1 `upc_all_file_sync` ensures that any data that has been written to the file associated with `fd` but not yet transferred to the storage device is transferred to the storage device. It also ensures that subsequent file reads from any thread will see any previously written values (that have not yet been overwritten).

2 The function returns 0 on success. On error, it returns –1 and sets `errno` appropriately.

7.3.2.4 The `upc_all_seek` function

Synopsis

```
#include <upc.h>
#include <upc_io.h>

    upc_off_t upc_all_seek(upc_file_t *fd,
                           upc_off_t offset,
```

```
                                    int origin)
```

Description

1 `upc_all_seek` sets the current position of the file pointer associated with `fd`.

2 This offset can be relative to the current position of the file pointer, to the beginning of the file, or to the end of the file. The offset can be negative, which allows seeking backwards.

3 The origin parameter can be specified as `UPC_SEEK_SET`, `UPC_SEEK_CUR`, or `UPC_SEEK_END`, respectively, to indicate that the offset must be computed from the beginning of the file, the current location of the file pointer, or the end of the file.

4 In the case of a shared file pointer, all threads must specify the same offset and origin. In the case of a private file pointer, each thread may specify a different offset and origin.

5 It is legal to seek past the end of file. It is erroneous to seek to a negative position in the file. See the Common Constraints number 5 at the beginning of Section 7.3 for more details.

6 The current position of the file pointer can be determined by calling `upc_all_seek(fd, 0, UPC_SEEK_CUR)`.

7 On success, the function returns the current location of the file pointer in bytes. If there is an error, it returns –1 and sets `errno` appropriately.

7.3.2.5 The `upc_all_file_set_size` function

Synopsis

```
#include <upc.h>
#include <upc_io.h>

    int upc_all_file_set_size(upc_file_t *fd,
                              upc_off_t size)
```

Description

1 `upc_all_file_set_size` executes an implicit `upc_all_file_sync` and resizes the file associated with `fd`.

2 `size` is measured in bytes from the beginning of the file.

3 If `size` is less than the current file size, the file is truncated at the position defined by `size`. The implementation is free to deallocate file blocks located beyond this position.

4 If `size` is greater than the current file size, the file size increases to `size`. Regions of the file that have been previously written are unaffected. The values of data in the new regions in the file (between the old size and `size`) are undefined.

5 If this function truncates a file, it is possible that the private and shared file pointers may point beyond the end of file. This is legal and is equivalent to seeking past the end of file (see the Common Rules in Section 5 for the semantics of seeking past the end of file).

6 It is implementation dependent whether this function allocates file space. Use `upc_all_file_preallocate` to force file space to be reserved.

7 Calling this function does not affect the private or shared file pointers.

8 The function returns 0 on success. On error, it returns –1 and sets `errno` appropriately.

7.3.2.6 The `upc_all_file_get_size` function

Synopsis

```
#include <upc.h>
#include <upc_io.h>

    upc_off_t upc_all_file_get_size(upc_file_t *fd)
```

Description

1 `upc_all_file_get_size` returns the current size in bytes of the file associated with `fd` on success. On error, it returns –1 and sets `errno` appropriately.

7.3.2.7 The `upc_all_file_preallocate` function

Synopsis

```
#include <upc.h>
#include <upc_io.h>

    int upc_all_file_preallocate(upc_file_t *fd,
                                 upc_off_t size)
```

Description

1 `upc_all_preallocate` ensures that storage space is allocated for the first `size` bytes of the file associated with `fd`.

2 Regions of the file that have previously been written are unaffected. For newly allocated regions of the file, `upc_all_file_preallocate` has the same effect as writing undefined data.

3 If `size` is greater than the current file size, the file size increases to `size`. If `size` is less than or equal to the current file size, the file size is unchanged.

4 Calling this function does not affect the private or shared file pointers.

5 The function returns 0 on success. On error, it returns –1 and sets `errno` appropriately.

7.3.2.8 The `upc_all_fcntl` function

Synopsis

```
#include <upc.h>
#include <upc_io.h>

    int upc_all_fcntl(upc_file_t *fd,
                      int cmd,
                      void *arg)
```

Description

1 `upc_all_fcntl` performs one of various miscellaneous operations related to the file specified by `fd`, as determined by `cmd`. The valid commands `cmd` and their associated argument `arg` are explained below.

`UPC_GET_CA_SEMANTICS`	Get the current consistency and atomicity semantics

	used. The argument `arg` is ignored. The return value is UPC_STRICT for strict consistency and atomicity semantics and 0 for the default weak consistency and atomicity.
`UPC_SET_WEAK_CA_SEMANTICS`	Executes an implicit `upc_all_file_sync` on `fd` and sets the weak consistency and atomicity semantics to be used from now on. The argument `arg` is ignored. The return value is 0 on success. On error, this function returns -1 and sets `errno` appropriately.
`UPC_SET_STRONG_CA_SEMANTICS`	Executes an implicit `upc_all_file_sync` on `fd` and sets the strong consistency and atomicity semantics to be used from now on. The argument `arg` is ignored. The return value is 0 on success. On error, this function returns -1 and sets `errno` appropriately.
`UPC_SET_SHARED_FP`	Executes an implicit `upc_all_file_sync` on `fd`, switches the current file access pointer mechanism to a shared file pointer, and seeks to the beginning of the file. The argument `arg` is ignored. The return value is 0 on success. On error, this function returns -1 and sets `errno` appropriately.
`UPC_SET_PRIVATE_FP`	Executes an implicit `upc_all_file_sync` on `fd`, switches the current file access pointer mechanism to a private file pointer, and seeks to the beginning of the file. The argument `arg` is ignored. The return value is 0 on success. On error, this function returns -1 and sets `errno` appropriately.
`UPC_GETFP`	Get the type of the current file pointer. The argument `arg` is ignored. The return value is either `UPC_SHARED_FP` in case of a shared file pointer, or `UPC_PRIVATE_FP` for private file pointers.
`UPC_GETFL`	Get the flags specified during the `upc_all_open` call. The argument `arg` is ignored. The return value has similar format to the `flags` parameter in `upc_all_open`.
`UPC_GETFN`	Get the file name provided by each thread in the `upc_all_open` call that created `fd`. The argument `arg` is a valid (`char**`) pointing to (`char*`) location in which a pointer to file name will be written. Writes a (`char*`) into `*arg` pointing to the filename in

	implementation-maintained read-only memory, which will remain valid until the file handle is closed or until the next `upc_all_fcntl` call on that file handle.
`UPC_GET_HINTS`	Retrieve the hints applicable to `fd`. The argument `arg` is a valid (`upc_hint**`) pointing to a (`upc_hint*`) location in which a pointer to the hints array will be written. Writes a (`upc_hint_t*`) into `*arg` pointing to an array of `upc_hint_t`'s in implementation-maintained read-only memory, which will remain valid until the file handle is closed or until the next `upc_all_fnctl` call on that file handle. The number of hints in the array is returned by the call. The hints in the array may be a subset of those specified at file open time, if the implementation ignored some unrecognized or unsupported hints.
`UPC_SET_HINT`	Executes an implicit `upc_all_file_sync` on `fd` and sets a new hint to `fd`. The argument `arg` points to one single-valued `upc_hint_t` hint to be applied. The return value is 0 on success. On error, this function returns -1 and sets `errno` appropriately.
`UPC_ASYNC_OUTSTANDING`	Returns 1 if there is an asynchronous operation outstanding for this file handle, or 0 otherwise.

2 In case of a non valid `fd`, `upc_all_fcntl` returns -1 and sets `errno` appropriately.

3 It is legal to call `upc_all_fcntl(UPC_ASYNC_OUTSTANDING)` when an asynchronous operation is outstanding (but it is still illegal to call `upc_all_fcntl` with any other argument when an asynchronous operation is outstanding).

7.3.3 Reading Data

Common Constraints

1 No function in this section (7.3.3) may be called while an asynchronous operation is pending on the file handle.

2 No function in this section (7.3.3) implies the presence of barriers at entry or exit. However, the programmer is advised to use a barrier after calling `upc_all_read_shared` to ensure that the entire shared buffer has been filled up.

7.3.3.1 The `upc_all_read_private` function

Synopsis

```
#include <upc.h>
#include <upc_io.h>

    ssize_t upc_all_read_private(upc_file_t *fd,
                                 void *buffer,
                                 size_t size)
```

Description

1 `upc_all_read_private` reads data from a file into a private buffer on each thread.

2 This function can be called only if the file was opened for reading with private file pointers. It is erroneous to call this function if the file was opened with the `UPC_SHARED_FP` flag.

3 `buffer` is a pointer to an array into which data will be read, and each thread may pass a different value for `buffer`.

4 `size` is the number of bytes that *each thread* must read from the file, and each thread may pass a different value for `size`. `size` may be zero, in which case the `buffer` argument is ignored and that thread performs no I/O.

5 On success, the function returns the number of bytes read into the private buffer of the calling thread, and the private file pointer of the thread is incremented by that amount. On error, it returns –1 and sets `errno` appropriately.

7.3.3.2 The `upc_all_read_shared` function

Synopsis

```
#include <upc.h>
#include <upc_io.h>

    ssize_t upc_all_read_shared(upc_file_t *fd,
                                shared void *buffer,
                                size_t blocksize,
```

```
                                        size_t size)
```

Description

1 `upc_all_read_shared` reads data from a file into a shared buffer in memory.

2 The function can be called either when the file was opened for reading with a shared file pointer or when the file was opened for reading with private file pointers.

3 `buffer` is a pointer to an array into which data will be read. It must be a pointer to shared data and may have affinity to any thread. `blocksize` is the block size of the shared buffer in bytes. A `blocksize` of 0 indicates an indefinite blocking factor.

4 In the case of private file pointers, the following rules apply: Each thread may pass a different address for the `buffer` parameter. `size` represents the number of bytes that each thread must read from the file. Each thread may specify a different value for `size` and may read from a different location in the file as specified by its private file pointer. `size` may be zero, in which case the `buffer` argument is ignored and that thread performs no I/O. On success, the function returns the number of bytes read by the calling thread, and the private file pointer of the thread is incremented by that amount.

5 In the case of a shared file pointer, the following rules apply: All threads must pass the same address for the `buffer` parameter. `size` represents the total number of bytes that all threads read into the shared buffer. All threads must pass the same value for `size`. `size` may be zero, in which case the `buffer` argument is ignored and the operation has no effect. On success, the function returns the total number of bytes read by all threads, and the shared file pointer is incremented by that amount.

6 If reading with private file pointers results in overlapping reads in the shared buffer, the result is determined by whether the file was opened with the `UPC_STRICT` flag or not (see Section 7.3.2.1).

7 The function returns –1 on error and sets `errno` appropriately.

7.3.4 Writing Data

Common Constraints

1 No function in this section (7.3.4) may be called while an asynchronous operation is pending on the file handle.

2 No function in this section (7.3.4) implies the presence of barriers at entry or exit. However, the programmer is advised to use a barrier before calling `upc_all_write_shared` to ensure that the entire shared buffer is up-to-date before being written to the file.

.

7.3.4.1 The `upc_all_write_private` function

Synopsis

```
#include <upc.h>
#include <upc_io.h>

    ssize_t upc_all_write_private(upc_file_t *fd,
                                  void *buffer, size_t size)
```

Description

1 `upc_all_write_private` writes data from a private buffer on each thread into a file.

2 This function can be called only if the file was opened for writing with private file pointers. It is erroneous to call this function if the file was opened with the `UPC_SHARED_FP` flag.

3 `buffer` is a pointer to an array from which data will be written, and each thread may pass a different value for `buffer`.

4 `size` is the number of bytes that each thread must write to the file, and each thread may pass a different value for `size`. `size` may be zero, in which case the `buffer` argument is ignored and that thread performs no I/O.

5 If any of the writes result in overlapping accesses in the file, the result is determined by whether the file was opened with the `UPC_STRICT` flag or not (see Section 7.3.2.1).

6 On success, the function returns the number of bytes written by the calling thread, and the private file pointer of the thread is incremented by that amount. On error, it returns –1 and sets `errno` appropriately.

7.3.4.2 The `upc_all_write_shared` function

Synopsis

```
#include <upc.h>
#include <upc_io.h>

    ssize_t upc_all_write_shared(upc_file_t *fd,
                                 shared void *buffer,
                                 size_t blocksize,
                                 size_t size)
```

Description

1 `upc_all_write_shared` writes data from a shared buffer in memory to a file.

2 The function can be called either when the file was opened for writing with a shared file pointer or when the file was opened for writing with private file pointers.

3 `buffer` is a pointer to an array from which data will be written. It must be a pointer to shared data and may have affinity to any thread. `blocksize` is the block size of the shared buffer in bytes. A `blocksize` of 0 indicates an indefinite blocking factor.

4 In the case of private file pointers, the following rules apply: Each thread may pass a different address for the `buffer` parameter. `size` represents the number of bytes that each thread must write to the file. Each thread may specify a different value for `size` and may write to a different location in the file as specified by its private file pointer. `size` may be zero, in which case the `buffer` argument is ignored and that thread performs no I/O. On success, the function returns the number of bytes written by the calling thread, and the private file pointer of the thread is incremented by that amount.

5 In the case of a shared file pointer, the following rules apply: All threads must pass the same address for the `buffer` parameter. `size` represents the total number of bytes that all threads write from the shared buffer. All threads must pass the same value for `size`. `size` may be zero, in which case the `buffer` argument is ignored and the operation has no effect. On success, the function returns the total number of bytes written by all threads, and the shared file pointer is incremented by that amount.

6 If writing with private file pointers results in overlapping accesses in the file, the result is determined by whether the file was opened with the `UPC_STRICT` flag or not (see Section 7.3.2.1).

7 The function returns –1 on error and sets `errno` appropriately.

7.3.5 List I/O

Common Constraints

1 UPC-IO functions are collective and must be called by all threads. (See Section 3.6 of the UPC Specification [1] for the definition of collective).

2 List I/O functions take a list of addresses/offsets and corresponding lengths in memory and file to read from or write to.

3 List I/O functions can be called regardless of whether the file was opened for private file pointers or the shared file pointer.

4 File pointers are not updated as a result of a list I/O read/write operation.

5 The `memvec` argument passed to any list I/O *read* function by a single thread must not specify overlapping regions in memory.

6 The base addresses passed to `memvec` can be in any order.

7 The `filevec` argument passed to any list I/O *write* function by a single thread must not specify overlapping regions in the file.

8 The offsets passed in `filevec` must be in monotonically non-decreasing order.

9 No function in this section (7.3.5) may be called while an asynchronous operation is pending on the file handle.

10 No function in this section (7.3.5) implies the presence of barriers at entry or exit. However, the programmer is advised to use a barrier after calling `upc_all_read_list_shared` to ensure that the entire shared buffer has been filled up, and similarly, use a barrier before calling `upc_all_write_list_shared` to ensure that the entire shared buffer is up-to-date before being written to the file.

Note that for all the list I/O functions, each thread passes an independent set of memory and file vectors. Passing the same vectors on two or more threads specifies redundant work.

7.3.5.1 The `upc_all_read_list_private` function

Synopsis

```
#include <upc.h>
#include <upc_io.h>

    ssize_t upc_all_read_list_private(upc_file_t *fd,
                      size_t memvec_entries,
                      upc_private_memvec_t const *memvec,
                      size_t filevec_entries,
                      upc_filevec_t const *filevec)
```

Description

1 `upc_all_read_list_private` reads data from a file that was opened for reading into private buffers in memory.

2 `memvec_entries` indicates the number of entries in the array `memvec` and `filevec_entries` indicates the number of entries in the array `filevec`. The values may be 0, in which case the `memvec` or `filevec` argument is ignored and no locations are specified for I/O.

3 The result is as if data were read in order from the list of locations specified by `filevec` and placed in memory in the order specified by the list of locations in `memvec`. The total amount of data specified by `memvec` must equal the amount of data specified by `filevec`.

4 On success, the function returns the number of bytes read by the calling thread. On error, it returns –1 and sets `errno` appropriately.

7.3.5.2 The `upc_all_read_list_shared` function

Synopsis

```
#include <upc.h>
#include <upc_io.h>

    ssize_t upc_all_read_list_shared(upc_file_t *fd,
                      size_t memvec_entries,
                      upc_shared_memvec_t const *memvec,
                      size_t filevec_entries,
                      upc_filevec_t const *filevec)
```

Description

1 `upc_all_read_list_shared` reads data from a file that was opened for reading into various locations of a shared buffer in memory.

2 `memvec_entries` indicates the number of entries in the array `memvec` and `filevec_entries` indicates the number of entries in the array `filevec`. The values may be 0, in which case the `memvec` or `filevec` argument is ignored and no locations are specified for I/O.

3 The result is as if data were read in order from the list of locations specified by `filevec` and placed in memory in the order specified by the list of locations in `memvec`. The total amount of data specified by `memvec` must equal the amount of data specified by `filevec`.

4 If any of the reads from different threads result in overlapping regions in memory, the result is determined by whether the file was opened with the `UPC_STRICT` flag or not (see Section 7.3.2.1).

5 On success, the function returns the number of bytes read by the calling thread. On error, it returns –1 and sets `errno` appropriately.

6 *Note: With the above definition, there is no way to do with explicit offsets the equivalent of `upc_all_read_shared` using a shared file pointer, namely, where all threads specify the same access (same parameters), the data gets read collectively into the shared buffer, and the function returns the total amount of data read by all threads.*

7.3.5.3 The `upc_all_write_list_private` function

Synopsis

```
#include <upc.h>
#include <upc_io.h>

    ssize_t upc_all_write_list_private(upc_file_t *fd,
                    size_t memvec_entries,
                    upc_private_memvec_t const *memvec,
                    size_t filevec_entries,
                    upc_filevec_t const *filevec)
```

Description

1 `upc_all_write_list_private` writes data from private buffers in memory to a file that was opened for writing.

2 `memvec_entries` indicates the number of entries in the array `memvec` and `filevec_entries` indicates the number of entries in the array `filevec`. The values may be 0, in which case the `memvec` or `filevec` argument is ignored and no locations are specified for I/O.

3 The result is as if data were written from memory locations in the order specified by the list of locations in `memvec` to locations in the file in the order specified by the list in `filevec`. The total amount of data specified by `memvec` must equal the amount of data specified by `filevec`.

4 If any of the writes from different threads result in overlapping accesses in the file, the result is determined by whether the file was opened with the `UPC_STRICT` flag or not (see Section 7.3.2.1).

5 On success, the function returns the number of bytes written by the calling thread. On error, it returns –1 and sets `errno` appropriately.

7.3.5.4 The `upc_all_write_list_shared` function

Synopsis

```
#include <upc.h>
#include <upc_io.h>

    ssize_t upc_all_write_list_shared(upc_file_t *fd,
                size_t memvec_entries,
                upc_shared_memvec_t const *memvec,
                size_t filevec_entries,
                 upc_filevec_t const *filevec)
```

Description

1 `upc_all_write_list_shared` writes data from various locations of a shared buffer in memory to a file that was opened for writing.

2 `memvec_entries` indicates the number of entries in the array `memvec` and `filevec_entries` indicates the number of entries in the array `filevec`. The values may be 0, in which case the `memvec` or `filevec` argument is ignored and no locations are specified for I/O.

3 The result is as if data were written from memory locations in the order specified by the list of locations in `memvec` to locations in the file in the order specified by

the list in `filevec`. The total amount of data specified by `memvec` must equal the amount of data specified by `filevec`.

4 If any of the writes from different threads result in overlapping accesses in the file, the result is determined by the atomicity settings, namely, whether the file was opened with the `UPC_STRICT` flag or not (see Section 7.3.2.1).

5 On success, the function returns the number of bytes written by the calling thread. On error, it returns –1 and sets `errno` appropriately.

6 *Note: With the above definition, there is no way to do with explicit offsets the equivalent of `upc_all_write_shared` using a shared file pointer, namely, where all threads specify the same access (same parameters), the data gets written collectively from a shared buffer, and the function returns the total amount of data written by all threads.*

7.3.6 Asynchronous I/O

Common Constraints

1 Only one asynchronous I/O operation can be outstanding on a UPC-IO file handle at any time.

2 For asynchronous read operations, the contents of the destination memory are undefined until after a successful `upc_all_async_wait` or `upc_all_async_test` on the file handle. For asynchronous write operations, the source memory may not be safely modified until after a successful `upc_all_async_wait` or `upc_all_async_test` on the file handle.

3 If an error occurs during the completion of one asynchronous I/O operation, the position of the file pointer becomes undefined.

4 An implementation is free to block for completion of an operation in the asynchronous initiation call or in the `upc_all_async_test` call (or both). High-Quality implementations are recommended to minimize the amount of time spent within the asynchronous initiation or `upc_all_async_test` call.

5 In the case of list I/O functions, the user may modify or free the lists after the asynchronous I/O operation has been initiated.

7.3.6.1 The `upc_all_async_read_private` function

Synopsis

```
#include <upc.h>
#include <upc_io.h>

    void upc_all_async_read_private(upc_file_t *fd,
                                    void *buffer,
                                    size_t size)
```

Description

1 `upc_all_async_read_private` initiates an asynchronous read from a file into a private buffer on each thread.

2 The meaning of the parameters and restrictions are the same as for the blocking function, `upc_all_read_private`.

3 The status of this asynchronous I/O operation can be retrieved by calling `upc_all_async_test` or `upc_all_async_wait`.

7.3.6.2 The `upc_all_async_read_shared` function

Synopsis

```
#include <upc.h>
#include <upc_io.h>

    void upc_all_async_read_shared(upc_file_t *fd,
                                   shared void *buffer,
                                   size_t blocksize,
                                   size_t size)
```

Description

1 `upc_all_async_read_shared` initiates an asynchronous read from a file into a shared buffer on each thread.

2 The meaning of the parameters and restrictions are the same as for the blocking function, `upc_all_read_shared`.

3 The status of this asynchronous I/O operation can be retrieved by calling `upc_all_async_test` or `upc_all_async_wait`..

7.3.6.3 The `upc_all_async_write_private` function

Synopsis

```
#include <upc.h>
#include <upc_io.h>

    void upc_all_async_write_private(upc_file_t *fd,
                                     void *buffer,
                                     size_t size)
```

Description

1 `upc_all_async_write_private` initiates an asynchronous write from a private buffer on each thread to a file.

2 The meaning of the parameters and restrictions are the same as for the blocking function, `upc_all_write_private`.

3 The status of this asynchronous I/O operation can be retrieved by calling `upc_all_async_test` or `upc_all_async_wait`.

7.3.6.4 The `upc_all_async_write_shared` function

Synopsis

```
#include <upc.h>
#include <upc_io.h>

    void upc_all_async_write_shared(upc_file_t *fd,
                                    shared void *buffer,
                                    size_t blocksize,
                                    size_t size)
```

Description

1 `upc_all_async_write_shared` initiates an asynchronous write from a shared buffer to a file.

2 The meaning of the parameters and restrictions are the same as for the blocking function, `upc_all_write_shared`.

3 The status of this asynchronous I/O operation can be retrieved by calling `upc_all_async_test` or `upc_all_async_wait`.

7.3.6.5 The `upc_all_async_read_list_private` function

Synopsis

```
#include <upc.h>
#include <upc_io.h>

  void upc_all_async_read_list_private(upc_file_t *fd,
                   size_t memvec_entries,
                   upc_private_memvec_t const *memvec,
                   size_t filevec_entries,
                   upc_filevec_t const *filevec)
```

Description

1 `upc_all_async_read_list_private` initiates an asynchronous read of data from a file into private buffers in memory.

2 The meaning of the parameters and restrictions are the same as for the blocking function, `upc_all_read_list_private`.

3 The status of this asynchronous I/O operation can be retrieved by calling `upc_all_async_test` or `upc_all_async_wait`.

7.3.6.6 The `upc_all_async_read_list_shared` function

Synopsis

```
#include <upc.h>
#include <upc_io.h>

   void upc_all_async_read_list_shared(upc_file_t *fd,
                   size_t memvec_entries,
                   upc_shared_memvec_t const *memvec,
                   size_t filevec_entries,
                   upc_filevec_t const *filevec)
```

Description

1 `upc_all_read_list_shared` initiates an asynchronous read of data from a file into various locations of a shared buffer in memory.

2 The meaning of the parameters and restrictions are the same as for the blocking function, `upc_all_read_list_shared`.

3 The status of this asynchronous I/O operation can be retrieved by calling `upc_all_async_test` or `upc_all_async_wait`.

7.3.6.7 The `upc_all_async_write_list_private` function

Synopsis

```
#include <upc.h>
#include <upc_io.h>

  void upc_all_async_write_list_private(upc_file_t *fd,
                    size_t memvec_entries,
                    upc_private_memvec_t const *memvec,
                    size_t filevec_entries,
                    upc_filevec_t const *filevec)
```

Description

1 `upc_all_async_write_list_private` initiates an asynchronous write of data from private buffers in memory to a file.

2 The meaning of the parameters and restrictions are the same as for the blocking function, `upc_all_write_list_private`.

3 The status of this asynchronous I/O operation can be retrieved by calling `upc_all_async_test` or `upc_all_async_wait`.

7.3.6.8 The `upc_all_async_write_list_shared` function

Synopsis

```
#include <upc.h>
#include <upc_io.h>

  void upc_all_async_write_list_shared(upc_file_t *fd,
                    size_t memvec_entries,
                    upc_shared_memvec_t const *memvec,
                    size_t filevec_entries,
                    upc_filevec_t const *filevec)
```

Description

1 `upc_all_async_write_list_shared` initiates an asynchronous write of data from various locations of a shared buffer in memory to a file.

2 The meaning of the parameters and restrictions are the same as for the blocking function, `upc_all_write_list_shared`.

3 The status of this asynchronous I/O operation can be retrieved by calling `upc_all_async_test` or `upc_all_async_wait`.

7.3.6.9 The `upc_all_async_wait` function

Synopsis

```
#include <upc.h>
#include <upc_io.h>

    ssize_t upc_all_async_wait(upc_file_t *fd)
```

Description

1 `upc_all_async_wait` completes the previously issued asynchronous I/O operation on the file handle `fd`.

2 It is erroneous to call this function if there is no outstanding asynchronous I/O operation associated with `fd`.

3 On success, the function returns the number of bytes read or written by the asynchronous I/O operation as specified by the blocking variant of the function used to initiate the asynchronous operation. On error, it returns –1 and sets `errno` appropriately, and the outstanding asynchronous operation (if any) becomes no longer outstanding.

7.3.6.10 The `upc_all_async_test` function

Synopsis

```
#include <upc.h>
#include <upc_io.h>

    ssize_t upc_all_async_test(upc_file_t *fd,
                               int *flag)
```

Description

1 `upc_all_async_test` tests whether the outstanding asynchronous I/O operation associated with `fd` has completed.

2 If the operation has completed, the function sets `flag=1` and the asynchronous operation becomes no longer outstanding[‡]; otherwise it sets `flag=0`. The same value of `flag` is returned on all threads.

3 If the operation was completed, the function returns the number of bytes that were read or written as specified by the blocking variant of the function used to initiate the asynchronous operation. On error, it returns –1 and sets `errno` appropriately, and sets the `flag=1`, and the outstanding asynchronous operation (if any) becomes no longer outstanding.

4 It is erroneous to call this function if there is no outstanding asynchronous I/O operation associated with `fd`.

[‡] This implies it is illegal to call `upc_all_async_wait` or `upc_all_async_test` immediately after a successful `upc_all_async_test` on that file handle.

References

[1] Tarek A. El-Ghazawi, William W. Carlson, and Jesse M. Draper. UPC Language Specifications V1.1 (http://upc.gwu.edu). March 2003.

[2] William W. Carlson, Jesse M. Draper, David Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. Introduction to UPC and Language Specification CCS-TR-99-157.

[3] American National Standards Institute, American National Standard for Information Systems, Programming Language--C, 1989.

[4] Tarek A. El-Ghazawi, F. Cantonnet, UPC Performance and Potential: A NPB Experimental Study Supercomputing2002 (SC2002), IEEE, Baltimore, November 2002

[5] Tarek A. El-Ghazawi, Programming in UPC (ftp://ftp.seas.gwu.edu/pub/upc/downloads/tut/sld001.htm), March 2001.

[6] MPI-2: Extensions to the Message-Passing Interface, Message Passing Interface Forum, July 18, 1997.

[7] John M. May, Parallel I/O for High Performance Computing, Morgan Kaufmann, October 2000.

[8] Intel Corporation, Paragon System User's Guide, May 1995, 312489-004

[9] ISO Programming Languages-C. ISO/SEC 9899. May 2000.

[10] Elizabeth Wiebel, David Greenberg and Steven Seidel, UPC Collective Operations Specification V1.0 pre4 (ftp://ftp.seas.gwu.edu/pub/upc/downloads/Coll-pre4-V1-0.ps), April 2003.

APPENDIX A : Open Issues

This section describes features that will be discussed in future releases of the UPC-IO Specifications.

1. Returning error values and errno are a good idea, especially since there's a number of I/O errors (e.g. "file not found") which some applications may want to treat as non-fatal. However, doing so requires that we specify what the possible error values are and ideally which functions may return which errors. Can we just use the standard POSIX provided errno values, or do we need additional ones to handle UPC-IO specific errors? (if so, then it's probably not appropriate to be using POSIX errno, which has connections to system-defined functions like perror() and strerror()).

2. We may need to rename some identifiers to accommodate recent terminology changes in the UPC language specification. For example, we may need to rename `upc_private_memvec_t` to `upc_local_memvec_t` and the `*_private` functions to `*_local` (although we can keep the “private” terminology for file pointers, which may actually improve clarity by not overloading terms).

3. Everything needs to be converted to LaTeX for merging with language spec. We need to certain not to lose any wording, etc during this conversion.

4. We may want to provide single-valued errors for `upc_all_open` and `upc_all_close` – otherwise the state of the file handle is indeterminate after an error (i.e. all threads should always agree upon whether the file is officially open or not). The performance penalty for this should be minimal, since file open/close should be relatively infrequent and barrier synchronization is probably required here anyhow.

5. We may want to add support for allowing multiple outstanding asynchronous operations on the same file handle. This can be done by introducing a handle datatype like `upc_all_handle_t` to represent the explicitly non-blocking collective operation in flight, and have this value returned by each async init function and consumed by `upc_all_async_test()`/`upc_all_async_wait()`. We may even want to use the same datatype and sync functions or explicitly non-blocking collectives.

6. The proposed collective communication functions now take an extra sync_mode argument. Assuming that it is approved, we may want to add something similar to UPC-IO data movement operations to allow explicit control of the barrier semantics.